**COEN 174 Software Engineering Fall 2021**

**Lab 7 (60 pts)**

**Stephen Tambussi**

**In this lab, you will use Eclipse IDE to**

* How to auto-generate constructors, setters and getters in a Java Class
* How to **refactor code** to
* extract a method
* extract a superclass
* extract interface
* move a method
* Implement template method design pattern

**Exercise 1 (10 pts)**

In this exercise, you will define a class and **auto generate the constructors, setters and getters for some of the attributes.**

* Start Eclipse IDE. Create a new Project called **COEN174Lab7**. Inside the project, create a package called **edu.coen174.lab7.ex123**
* Underneath this package create a file called **Exercise123.java**. Inside that file create a public class called **Exercise123** (with main) and add the code below.

Note: For sake of convenience, the classes Employee, PartTimeEmployee and Organization are defined in the same file as class Exercise1, since they are not public classes. You are free to define them in their own files as public classes.

**class** Employee {

**private** String name;

**private** String empId;

**private** **double** salary;

**private** **int** hrsWorked;

**private** **double** billingRate;

**private** String skill;

}

* From the toolbar, click on the Source option (as shown below) and select Generate Constuctor using Fields.



Figure 1

You will see the options below. Now, choose the *name* and *empId* fields to initialize in the constructor and choose the option *to Omit call to default superclass constructor* ….. Click **Generate** to generate a constructor for class Employee.



* Add (manually) a default constructor with no args and leave the body empty.
* Use the option to **generate Getters and Setters** (see Fig1) and generate a setter and a getter for the attribute, *salary*.
* Use the option to generate **toString()** (see Fig1) with attributes, **name** and **empId**
* You may want to test the constructor and the toString(). In main() of Exercise1 class,

Employee emp1 = **new** Employee("give a name of your choice","id of your choice");

System.***out***.println(emp1); // This calls the toString()

**Exercise 2 (10 pts)**

Define a class called **PartTimeEmployee**.

**class** PartTimeEmployee **extends** Employee {

**private** **double** hrlyPay;

}

At this point, from analysis of Employee class, it is learned that the attributes, hrsWorked, billingRate and skill are applicable only for PartTimeEmployees. Now, you want to **refactor the code in Employee** class as follows: Select the three attributes in Employee class and choose **push down** as shown in Fig:3.

See Fig 3. From the refactor option (from the main tool bar)



Fig: 3

Now, your PartTimeEmployee class should have the 4 attributes, hrlyPay, hrsWorked, billingRate and skill.

Autogenerate two constructors for class PartTimeEmployee – one a default constructor and the other to initialize all the data members (including superclass).

Autogenerate a getter for each of hrlyPay, hrsWorked, billingRate and skill.



**Exercise 3 (20 pts)**

Define a class called **Organization** as shown below:

**class Organization {**

**private String orgName;**

**private double billingRate;**

**private String skill;**

**}**

* Auto-generate a constructor to initialize all the data members.
* Auto-generate getters for all data members.

At this point, we realize the common methods, getBillingRate() and getSkill() in classes **PartTimeEmployee** and **Organization**. Let us extract these two methods and create an **interface** called **Billable**. Click on class **Organization** and choose **Extract Interface** and name the interface **Billable.**



**Fig: 4**

**Eclipse will create the interface Billable (in its own file) and will automatically include the implements Billable in class Organization.** Manually, insert “implements Billable” in class PartTimeEmployee as well.

**Testing:**

You should test your classes now. Include the code below in the **main() of Exercise123** class and run your program.

**Billable partEmp = new PartTimeEmployee("D.Smith", "123",20.00, 30, 10.00,"programming");**

**Billable org = new Organization("ABCOrg", 100.00,"ProjectManagement");**

**System.*out*.println(partEmp.getSkill());**

**System.*out*.println(org.getSkill());**

Can you call any other methods in partEmp and org objects? If not, why not?

You can call getBillingRate() in addition to getSkill(), but no other methods other than these two because it is type billable and not type org or emp.

**Exercise 4 (20 pts)**

Examine the classes below:

**class** **OnlineOrder**{

**public** **void** processOrder(**boolean** giftWrap) {

// Select Item

System.***out***.println("Select the item");

System.***out***.println("Select the delivery address");

// Make Payment

System.***out***.println("Make payment by Credit card");

// Check if giftwrap is required

**if** (giftWrap)

System.***out***.println("Will be gift-wrapped");

// deliverOrder

System.***out***.println("Order will be delivered on the shown date");

}

}

**class** **InStoreOrder** {

**public** **void** processOrder(**boolean** giftWrap) {

// Select Item

System.***out***.println("Item scanned at the Register");

// Make Payment

System.***out***.println("Make payment by Cash or Credit card");

// Check if giftwrap is required

**if** (giftWrap)

System.***out***.println("Will be gift-wrapped");

// deliverOrder

System.***out***.println("Order delivered to the Customer at the Counter");

}

}

**Create a package called edu.coen174.lab7.ex4 and inside create a class called Exercise4 with main(). Add the above classes into the same file.**

You will notice that both classes have a ***processOrder()*** method which does 4 things:

* **Select item** b) **Make payment** c) Check if giftwrap is selected and 4) Deliver the Order.
* But the details of each of these steps are different. Note: Only print statements are shown to keep the code simple. In reality, there will be many lines of code for each step.
* **Since the method processOrder() is doing these four steps, each step can be made into a method.** We will use the refactoring, **ExtractMethod** to create 4 methods.

**ExtractMethod Refactoring**

In class **OnLineOrder**, highlight the code that is shown (highlighted) and choose ExtractMethod option from the Refactor option on the toolbar. You should extract one method at a time. So, highlight the two lines in the first highlighted section, and then the next highlighted code and so on.

You will be asked to select a name for the method. Choose **selectItem** as the method name and make it a public method**.**

Similarly, select the code circled in the other segments. Give **makePayment(),** **okToGiftWrap** and **deliverOrder** as the names for the other methods.

**class** **OnlineOrder**{

**public** **void** processOrder(**boolean** giftWrap) {

// Select Item

System.***out***.println("Select the item");

System.***out***.println("Select the delivery address");

// Make Payment

System.***out***.println("Make payment by Credit card");

// Check if giftwrap is required

**if** (giftWrap)

System.***out***.println("Will be gift-wrapped");

// deliverOrder

System.***out***.println("Order will be delivered on the shown date");

}

}

* Do the same to create methods (use same names)in class **InStoreOrder**.

**Extract SuperClass Refactoring**

Since the two classes, **OnLineOrder, InStoreOrder** have very similar functionality, let us first create a superclass from them.

Click and highlight the class name **OnLineOrder** and from the refactor option on the toolbar, choose **ExtractSuperClass.**.

You will be asked to enter the **name** of the super class (choose **CustOrder**) and you will be shown the list of methods in OnLineOrder that you want to move into the superclass. For now, do not

You will see now the class declaration is c**lass** OnlineOrder **extends** CustOrder and Eclipse now would have created a public class CustOrder in the file, CustOrder.java.

Open the file (from the PackageExplorer pane on the left side) CustOrder.

The class definition is empty at this point. Manually, add the highlighted code as shown below.

**public** **abstract** **class** CustOrder {

**public** **abstract** **void** selectItem();

**public** **abstract** **void** makePayment();

**public** **abstract** **void** okToGiftWrap(**boolean** giftWrap);

**public** **abstract** **void** deliverOrder();

}

Now, change the declaration of class InStoreOrder as

**class** InStoreOrder **extends** CustOrder

**Important Note and the idea central to this set of refactorings:**

If you see the processOrder() method, the steps in that method (4 method calls) should always be done in the same sequence, in any of the subclasses. The only details that will be different will be the code in the 4 methods, *selectItem(), makePayment(), okToGiftWrap()* and *deliverOrder().*

We will do a refactoring, **pullup,** to move the method, processOrder() to superclass, CustOrder.

Highlight the method ***processOrder()*** in **OnLineOrder** class and choose pullup option in refactoring. This will effectively move processOrder into superclass, CustOrder.

**Now, remove, processOrder() from InStoreOrder class (since it will be inherited from the superclass).**

**Go to class CustOrder and make the method, processOrder() final (it cannot be overridden by the subclasses).**

After your refactorings, your final code should look as below.

**public** **abstract** **class** CustOrder {

**final** **public** **void** processOrder(**boolean** giftWrap) {

selectItem();

makePayment();

okToGiftWrap(giftWrap);

deliverOrder();

}

**public** **abstract** **void** selectItem();

**public** **abstract** **void** makePayment();

**public** **abstract** **void** okToGiftWrap(**boolean** giftWrap);

**public** **abstract** **void** deliverOrder();

}

**class** OnlineOrder **extends** CustOrder {

**public** **void** deliverOrder() {

// deliverOrder

System.***out***.println("Order will be delivered on the shown date");

}

**public** **void** okToGiftWrap(**boolean** giftWrap) {

// Check if giftwrap is required

**if** (giftWrap)

System.***out***.println("Will be gift-wrapped");

}

**public** **void** makePayment() {

// Make Payment

System.***out***.println("Make payment by Credit card");

}

**public** **void** selectItem() {

// Select Item

System.***out***.println("Select the item");

System.***out***.println("Select the delivery address");

}

}

**class** InStoreOrder **extends** CustOrder{

**public** **void** deliverOrder() {

// deliverOrder

System.***out***.println("Order delivered to the Customer at the Counter");

}

**public** **void** okToGiftWrap(**boolean** giftWrap) {

// Check if giftwrap is required

**if** (giftWrap)

System.***out***.println("Will be gift-wrapped");

}

**public** **void** makePayment() {

// Make Payment

System.***out***.println("Make payment by Cash or Credit card");

}

**public** **void** selectItem() {

// Select Item

System.***out***.println("Item scanned at the Register");

}

}

**Time to test our code.**

Include the code statements below in the main() and run your program.

CustOrder order1 = **new** OnlineOrder();

order1.processOrder(**true**);

CustOrder order2 = **new** InStoreOrder();

order2.processOrder(**false**);

You have implemented a DesignPattern called TemplateMethod Pattern in this exercise. We will discuss Design Patterns in class.